

标下划线的为重点

一、C与C++

1.C和C++区别：

(1)**C是面向过程编程**，C++是面向对象编程(三大特性)

(2)**C是C++的子集**，C++可以兼容C语言的特性，还增加了一些新特性，比如auto变量、引用、智能指针等

(3)**C语言有一些不安全的特性**，比如指针使用潜在的危险性、强制类型转换的不确定性，C++增加了一些特性来改善安全问题，比如C++中有引用、智能指针、cast转换、异常捕获机制try-catch等

指针使用危险性：指针最初指向一个对象，之后又为这个指针分配其他对象，而原先对象没有释放造成内存泄漏。悬空指针（指针的指向对象已被删除）和野指针（指针变量在定义时如果未初始化）可能会去访问不可控内存空间，破坏正常的的数据，引发未知错误。

强制类型转换的不确定性:比如char* 转 int*会导致取得的值很奇怪,再有int和指针之间的转换是风险极高的一种转换，一个具体的地址赋值给指针变量是非常危险的，因为该地址上的内存可能没有分配，也可能没有读写权限，一般会导致程序崩溃。

(4)**C++代码的复用性高**，因为C++中有重载和模板这些概念，在此基础上，实现了方便开发的标准模板库STL

模板：有函数模板和类模板。

函数模板是建立一个通用函数，它所用到的数据的类型（包括返回值类型、形参类型、局部变量类型）可以不具体指定，用一个虚拟的类型来代替，等发生函数调用时再根据传入的实参的具体类型来逆推出真正的类型。

类模板就是建立一个通用类，它的成员变量类型和成员函数的返回值、形参类型不具体指定，用一个虚拟的类型来代替，使用类模板定义对象时，再根据实参类型来取代类模板中的虚拟类型。

重载：同一作用域下的同名函数，才存在重载关系。它的特点是函数名相同，参数的类型、数量或者顺序要有所不同，函数的返回值不能够作为函数重载的条件。

2.面向对象和面向过程：

面向过程:是根据业务逻辑来分析解决一个问题都需要哪些步骤，用函数把这些步骤依次实现。

面向对象:把构成问题的事物抽象成各个对象，把事物的属性变量和操作属性变量的函数封装进去，这样在写代码时以对象为核心，调用对象的成员方法就可以完成一些事情。

3.C和C++的结构体struct区别

- 1.C中结构体没有成员函数和静态成员，C++中结构体可以有
- 2.C中结构体中成员默认的访问权限是public且不可更改，C++中也是public但可以更改为protected、private
- 3.C中结构体不可以继承，C++中结构体可以从类或其他结构体继承
- 4.C中结构体不能直接初始化数据成员，C++中可以
- 5.C中结构体在使用时要加上struct关键字，或用typedef取别名，C++可以省略struct关键字，直接使用

★4.简述C++从代码到可执行二进制文件的过程(动静态链接)

预编译（处理）、编译、汇编、链接。

1. 预编译：这个过程主要的处理操作如下： `gcc -E hello.c -o hello.i`

(1) 将所有的`#define`删除，并且展开所有的宏定义

(2) 处理所有的条件预编译指令，如`#if`、`#ifdef`

(3) 处理`#include`预编译指令，将被包含的文件插入到该预编译指令的位置。

(4) 过滤所有的注释

(5) 添加行号和文件名标识。

2. 编译：这个过程主要的处理操作如下： `gcc -S hello.i -o hello.s`

(1) 词法分析：将源代码的字符序列分割成一系列的记号。

(2) 语法分析：对记号进行语法分析，产生语法树。

(3) 语义分析：判断表达式是否有意义。

(4) 代码优化：

(5) 目标代码生成：生成汇编代码。

(6) 目标代码优化：

3. 汇编：这个过程主要是将汇编代码转变成机器可以执行的指令。

`gcc -c hello.s -o hello.o`

4. 链接：将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。 `gcc hello.o -o hello.out`

链接分为静态链接和动态链接。

静态链接，是在链接的时候就已经把要调用的函数或者过程链接到了生成的可执行文件中，就算你在去把静态库删除也不会影响可执行程序的执行；生成的静态链接库，Windows下以`.lib`为后缀，Linux下以`.a`为后缀。

而动态链接，是在链接的时候没有把调用的函数代码链接进去，而是在执行的过程中，再去找要链接的函数，生成的可执行文件中没有函数代码，只包含函数的重定位信息，所以当你删除动态库时，可执行程序就不能运行。生成的动态链接库，Windows下以.dll为后缀，Linux下以.so为后缀。

5.main()执行前后的操作

(1)main()函数执行前会执行一些初始化的操作

设置栈指针(为栈分配相关的位置，用来放一些局部变量和其他数据)

初始化全局变量和静态变量(即data段的内容) (C)

对未设置初值的全局变量赋初值(数值型short, int, long等为0，指针为NULL等，即.bss段的内容)

在main之前还会调用全局对象的构造函数

给main()函数传递参数，argc, argv(第一个**argc表示参数的个数**；第二个参数是指向字符串的指针数组，其中**argv[0]为自身运行目录路径和程序名**，argv[1]指向第一个参数、argv[2]指向第二个参数)

(2)在main函数执行后会调用全局对象的析构函数；

6.++i和i++

1. **赋值顺序不同**：++ i 是先加后赋值；i ++ 是先赋值后加；++i和i++都是分两步完成的。
2. **效率不同**：后置++执行速度比前置的慢。
3. **i++ 不能作为左值，而++i 可以**：

```
int i = 0;
int *p1 = &(++i); //正确
int *p2 = &(i++); //错误
++i = 1; //正确
i++ = 1; //错误
```

4. 两者都不是原子操作。

7.说说const int *a, int const *a, const int a, int *const a, const int *const a分别是什么，有什么特点。

1. `const int a;` //指的是a是一个常量，不允许修改。
2. `const int *a;` //a指针所指向的内存里的值不变，即(*a)不变
3. `int const *a;` //同const int *a;
4. `int *const a;` //a指针所指向的内存地址不变，即a不变
5. `const int *const a;` //都不变，即(*a)不变，a也不变

8.简述C++有几种传值方式，之间的区别是什么？

传参方式有这三种：**值传递**、**引用传递**、**指针传递**

1. 值传递：形参即使在函数体内值发生变化，也不会影响实参的值；
2. 引用传递：形参在函数体内值发生变化，会影响实参的值；
3. 指针传递：在指针指向没有发生改变的前提下，形参在函数体内值发生变化，会影响实参的值；

值传递用于对象时，整个对象会拷贝一个副本，这样效率低；而引用传递用于对象时，不发生拷贝行为，只是绑定对象，更高效；指针传递同理，但不如引用传递安全。

9.四种类型转换运算符

C++中四种类型转换：static_cast, dynamic_cast, const_cast, reinterpret_cast。

1、const_cast

const_cast用来将const/volatile ['vɔ:lɒtl] 类型转换为非const/volatile ['vɔ:lɒtl] 类型。具体用法如下：

- (1)可将常量指针转化成非常量的指针，并且仍然指向原来的对象
- (2)可将常量引用转换成非常量的引用，也是仍然指向原来的对象
(常量对象或者是基本数据类型不允许转化为非常量对象，**只能通过指针和引用**来修改)

2、static_cast ['stæɪɪk]

static_cast是“静态转换”，它是在编译期间转换，转换失败会抛出编译错误，主要用于明确**隐式转换**，它**只支持低风险**的转换操作，不支持引用/指针，如：

- (1)C语言中原有的隐式转换，如short转int、int转double、const转非const、(向上转型)等；
- (2)void指针和具体类型指针间的转换，如void指针转int*、char转void*等；

不可用于高风险转换，如：

- (1)两个不同的具体类型指针之间的转换
- (2)整型和指针之间的互相转换
- (3)不同类型的引用之间的转换

`static_cast`还可用于类层次结构中父类和子类间的指针或引用的转换。进行**上行转换（把子类的指针或引用转换成父类表示）**是安全的，进行下行转换（把父类指针或引用转换成子类表示），子类可能包含的成员更多，占用内存更大，把父类转子类很可能访问到了父类不该访问的地方，会越界。由于`static_cast`没有动态类型检查，所以不安全，结果未知；

3、`dynamic_cast` [daɪ'næmɪk]

`dynamic_cast`是在类的继承层次间进行类型转换时使用，允许向上转型，也允许向下转型。因为向上转型子类转父类是安全的，所以使用`dynamic_cast`和`static_cast`的效果相同，使用`dynamic_cast`还有额外开销，所以向上转型使用`static_cast`就可以了；向下转型有可能不安全，因为父类有可能去访问子类扩展的成员，这就要借助RTTI机制进行检测，确定安全才能转换成功，**转型不成功会返回一个空指针**。使用`dynamic_cast`前提是：**父类必须要有虚函数**，因为RTTI运行时的类型检测是依赖于虚函数表的实现。

1. 常用于子类和父类之间的类型转换（指针/引用）
2. 有返回值，可以进行类型（安全）检查

RTTI:

编译器会把存在继承关系的类的类型信息用指针“连接”起来，形成一个继承链，`dynamic_cast`在程序运行过程中遍历这个继承链，如果遍历过程中遇到了要转换的目标类型，就能够转换成功，如果直到继承链的顶点（最顶层的基类）还没有遇到要转换的目标类型，就转换失败。

4、`reinterpret_cast` [ˌriːɪn'tɜːrprət]

`reinterpret_cast`和C语言显示强转本质相同。可以用在不同类型指针之间，不同类型引用之间，`int`和指针之间的转换等，但是因为是强转，所以转换后可能会出问题，尽量少用；

10.封装、继承、多态

(1)封装

将客观事物抽象成类，并把类的属性和行为按照不同的访问权限封装到类的里面，有三种访问权限:public、private和protected。

public作用域下的成员(包括成员函数和成员变量)，类内成员可以访问，也允许类外的对象访问。

private作用域下的成员(包括成员函数和成员变量)，只限于类内成员可以访问。

protected作用域下的成员(包括成员函数和成员变量)，类内成员和派生类成员都可以访问，但不允许类外的任何访问。

(2)继承

继承是我在创建一个新的类时可以先获取到现有类的成员函数和成员对象，然后在这个基础上再定义自己的新成员，(现有类是基类，新建类是派生类)。有三种继承方式：公有继承、私有继承和保护继承。基类的私有成员，无论派生类使用哪种继承方式都无法访问。

在公有继承(public)方式下：

基类中所有public和protected权限下的成员在派生类中仍为public和protected属性。基类中所有private权限下成员在派生类中不能访问。

在保护继承(protected)方式下：

基类中所有public和protected权限下的成员在派生类中都为protected属性。还是基类中所有private权限下成员在派生类中不能访问。

在私有继承(private)方式下：

基类中所有public和protected权限下的成员在派生类中都为private属性。

(3)多态

是指基类指针可以按照基类的方式去做事，也可以按照派生类的方式去做事，它有多种形态，多态可分为静态多态(也叫编译时的多态，地址早绑定)和动态多态(也叫运行时的多态，地址晚绑定)，静态多态主要是指对重载函数的调用，在编译时就能根据实参确定应该调用哪个函数，(因此叫编译时的多态)；动态多态和继承、虚函数等有关，在程序运行时才能确定是使用基类成员函数还是派生类成员函数，因此叫运行时的多态。

看这里多态是通过虚函数来实现的，有了虚函数，基类指针指向基类对象时就使用基类的成员（包括成员函数和成员变量），指向派生类对象时就使用派生类的成员。换句话说，基类指针可以按照基类的方式来做事，也可以按照派生类的方式来做事，它有多种形态，或者说有多种表现方式，我们将这种现象称为**多态**

多态原理：

一个类中如果包含了一个虚函数，在编译时会生成一个虚表，子类继承了父类的虚函数，每个子类也会生成一个自己的虚表，这样子类和父类在创建对象时，在对象的内存模型中，都会有一个虚表指针，指向类的虚表，类的所有对象共享这一个虚表，但类的每个对象都有一个属于自己的虚表指针，基类指针指向不同的对象去调用非静态成员函数时，会把这个对象的地址也就是this指针传递给对象的虚表指针，到虚表中找到对应的虚函数入口地址，然后执行实际被调用的虚函数。

11.继承中构造和析构顺序

基类构造函数》子类成员变量的构造函数》子类的构造函数

析构顺序与构造正好相反

12.友元函数作用及场景

作用：友元是不同类的成员函数之间或者类的成员函数与一般函数之间进行数据共享的这么一个机制把。通过友元，一个普通函数或另一个类中的成员函数可以访问类中的私有成员和保护成员。

使用场景：普通函数定义为友元函数，使普通函数能够访问类的私有成员。

13.定义和声明的区别

变量定义：用于为变量分配存储空间，还可为变量指定初始值。程序中，变量只能定义一次。

变量声明：不会为变量分配存储空间，用于向程序表明变量的类型和名字，程序中，可以声明多次。

14. ++i 和 i++

- 1、i++ 返回原来的值，++i 返回加1后的值。
 - 2、i++ 不能作为左值，而++i可以。
 - 3、i++前者是先赋值，然后再自增；++i后者是先自增，后赋值。
- ++i和i++都是分两步完成的。因为++i 是后面一步才赋值的，所以它能够当作一个变量进行级联赋值，++i =; a =b, 即++i是一个左值（可被寻址的值）； i++ 的后面一步是自增，不是左值。

二、关键字及运算符

1. C++中struct和class关键字的区别

(1)类中成员的默认访问权限是private，结构体中成员默认访问权限是public

(2)类默认的继承方式是私有继承，结构体默认的继承方式是公有继承

(3)类可以使用模板，结构体不可以(类模板:是对一批仅仅成员数据类型不同的类的抽象)

(4)class 关键字可以用于定义模板参数，就像 `typename`，而 `struct` 不能用于定义模板参数

2.extern

声明外部变量【在函数或者文件外部定义的全局变量】

extern"C"

C++支持函数重载，编译器编译函数的过程中会根据函数名和函数形参列表重新命名；但是C语言不支持函数重载，编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

所以当我们想用C++代码调用C语言代码时，要加上extern "C"，这样会告诉编译器这部分代码按C语言而不是C++的方式进行编译。

这个功能主要用在下面的情况：

1.C++代码调用C语言代码

2.在C++的头文件中使用(C++出现以前，很多代码都是C语言写的，而且很底层的库也是C语言写的)

3.在多个人协同开发时，可能有的人比较擅长C语言，而有的人擅长C++，这样的情况下也会有用到

★3.static关键字

1 static作用

- (1)static关键字修饰的变量存储在全局区，默认初始化为0
- (2)static修饰的函数或变量只能在本文件中使用
- (3)static修饰的成员变量或成员函数，它们都是类的一部分，这个类的所有对象都共享这个静态成员变量和静态成员函数。

2 (静态)全局变量/函数

- (1)静态全局变量和全局函数都是在全局区
- (2)静态全局变量(函数)的作用域仅限于当前文件，全局变量(函数)的作用域是整个源程序，包括其他源文件。

3 (静态)局部变量

- (1)静态局部变量在全局区，局部变量在栈区
 - (2)静态局部变量生命周期是整个程序运行结束，局部变量生命周期是随着函数的结束而结束
 - (3)定义静态局部变量默认初始化为0，定义局部变量默认初始化一个随机值
 - (4)静态局部变量只初始化一次，而局部变量每调用一次函数都会重新初始化。
- 静态局部变量特点：只初始化一次，整个程序结束后才被操作系统回收，只能在定义的函数内使用。

4 (静态)成员变量/函数

静态成员变量

- (1)static修饰的成员变量，存放在静态存储区(全局区)，该类的所有对象都共享同一个静态成员。
- (2)静态成员变量因为存储在全局区/静态区，所以在编译阶段就为它分配了内存
- (3)静态成员变量在类内声明，类外初始化

静态成员函数

- (1)静态成员函数也是存放在静态全局区，被该类的所有对象所共享
- (2)普通成员函数有 this 指针，可以访问类中的任意成员；而静态成员函数没有 this 指针，只能访问静态成员

★3.const关键字

1、修饰变量

const修饰变量表示阻止这个变量被修改，在定义该变量的时候要对其进行初始化

2、修饰指针

既可以指定指针本身为const，也可以指定其指向的数据为const，也可以指定都为const

3、修饰函数

const修饰函数形参，表示它是一个输入参数，在函数内部不能改变其值（这个形参可以接收const或非const的实参，接收后的形参均为const）

4、类

const成员函数可以访问非const、const成员，也可以访问const对象内所有数据成员；

非const成员函数可以访问非const、const成员，不可以访问const对象内所有数据成员；（const对象只能调用const函数）

const类型的成员变量因为只能在定义的时候初始化，所以只能在初始化列表中初始化，不能在类外初始化。

4.内联函数(inline)

请你说说内联函数，为什么使用内联函数？需要注意什么？：

1.(C++内联函数通常与类一起使用。)内联函数是在编译阶段，编译器会把这个函数的代码副本放到每个调用该函数的地方。如果想把一个函数定义为内联函数，函数名前要加关键字inline.

2.正常的函数调用是需要一定的时间开销，需要**寻址（函数入口地址）**，如果函数体本身很简单，函数调用的时间开销可能远大于函数体执行的开销，这个时候就比较适合使用内联函数。内联函数是以空间换时间，不需要寻址，它会在函数调用的地方直接用函数体进行替换。

3.但是如果函数体的代码很长，使用内联函数对内存资源消耗比较大，或者函数体内有循环，那么执行函数体内代码的时间就要比函数调用的时间长，这些情况就不适合使用内联函数了。

5.内联函数inline和宏函数define的区别

1.宏函数本质上不是函数，是预处理器用复用代码的方式代替函数调用，省去函数的出栈入栈过程，这样可以提高效率。而内联函数本质是一个函数，它一般都是函数体代码比较简单的函数。

2.宏函数是在预处理阶段，把所有宏名用宏体来替换(#define a a*b);内联函数在编译阶段，编译器在每个内联函数被调用的地方直接把内联函数的内容展开。

3.宏函数没有类型检查，无论对错都直接进行替换；内联函数在编译阶段会进行类型的检查。

6.const和define的区别

const用于定义常量；而define用于定义宏，而宏也可以用于定义常量。都用于常量定义时，它们的区别有：

1. const生效于编译的阶段；define生效于预处理阶段。
2. const定义的常量，在C语言中是存储在内存中、需要额外的内存空间的；define定义的常量，运行时是直接的操作数，并不会存放在内存中。
3. #define只是简单的字符串替换，没有类型检查。而const定义的常量有对应的数据类型，会进行类型检查，可以避免一些低级的错误。
4. const常量可以进行调试，define不能进行调试，因为在预编译阶段就已经替换掉了

7.const*和 *const的区别

//const* 是常量指针，*const 是指针常量

int const *a; //a指针所指向的内存里的值不变，即(*a)不变

int *const a; //a指针所指向的内存地址不变，即a不变

8. const和volatile、mutable

1.volatile ['vɒ:lətl]

volatile关键字修饰的变量，会让系统总是从它所在内存读取数据，因为cpu到寄存器读数据比到内存里读数据更快，在程序运行的时候，有些频繁使用的变量编译器会对它做优化，把这个变量装入寄存器里。在多线程下，如果两个线程都要去访问某一个共享变量，一个线程使用的是内存中的这个变量的值，另一个线程使用的是寄存器中的这个变量的值，内存和寄存器中这个变量的值可能不同，

就会造成执行错误，**所以多线程下要使用volatile来修饰那些被多个线程共用的变量。**

(在多线程下，线程A先将共享变量读入寄存器，对其执行加1操作并写回内存，然后线程B再将共享变量读入寄存器，加1后写回内存，这样是没有任何问题，问题在于线程是并发执行，可能线程A还未将累加后的数据写回内存，线程B就已经开始读取数据到寄存器，这样线程B会读到修改之前的旧数据)

2.const

(1)使用const,可以防止变量被改变，在定义const变量的时候，需要对它初始化，因为以后就没有机会再去改变它了；

(2)在函数声明时，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变它的值

(3)对于类的成员函数，如果指定为const类型了，就表明这是一个常函数，它不能修改类的成员变量，类的常对象只能访问类的常成员函数；

(4)const类型变量还可以通过类型转换符const_cast将const类型转换为非const类型

(5、6可不说)

(5)const成员函数可以访问非const对象的非const数据成员、const数据成员，也可以访问const对象内的所有数据成员.

(6)非const成员函数可以访问非const对象的非const数据成员、const数据成员，但不可以访问const对象的任意数据成员；

3.mutable

被mutable修饰的变量，将永远处于可变的狀態，即使在一个const函数中。我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成const的。但是，有些时候，我们需要在**const函数里面修改一些跟类状态无关的数据成员，那么这个函数就应该被mutable来修饰，并且放在函数后面关键字位置。**

9.C++中class和struct

- (1)类中成员的默认访问权限是private，结构体中成员默认访问权限是public
- (2)类默认的继承方式是私有继承，结构体默认的继承方式是公有继承
- (3)类可以使用模板，结构体不可以(类模板:是对一批仅仅成员数据类型不同的类的抽象)

10.C和C+的结构体struct区别

- 1.C中结构体没有成员函数和静态成员，C++中结构体可以有
- 2.C中结构体中成员默认的访问权限是public且不可更改，C++中也是public但可以更改为protected、private
- 3.C中结构体不可以继承，C++中结构体可以从类或其他结构体继承(默认公有继承)
- 4.C中结构体不能直接初始化数据成员，C++中可以
- 5.C中结构体在使用时要加上struct关键字，或用typedef取别名，C++可以省略struct关键字，直接使用

11.struct和union

- 1.联合体和结构体都是由若干个数据类型不同的数据成员组成。使用时，联合体只有一个有效的成员；而结构体所有的成员都有效。
- 2.对联合体的不同成员赋值，将会对覆盖其他成员的值，而对于结构体，当对不同成员赋值时，相互之间不影响。

12.大端小端及其判断

小端存储是指字节数据的低位存储在低地址

大端存储是指字节数据的低位存储在高地址

·主机大多数是小端数据

·网络字节序为大端数据

判断：可以根据联合体来判断该系统是大端还是小端。因为联合体变量总是从低地址存储

14 explicit消除隐式转换 [ɪk'splɪsɪt]

explicit关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以**显示的方式进行类型转换**，注意以下几点：

- explicit 关键字只能用于类内部的构造函数声明上
- explicit 关键字作用于单个参数的构造函数
- 被explicit修饰的构造函数的类，不能发生相应的隐式类型转换

15. override

override指定子类的这个虚函数是重写的父类的，这样如果你重写父类的虚函数的时候不小心把名字写错了的话编译是不会通过的。

16. final

当不希望某个类再被继承，或者不希望某个虚函数被重写，可以在类名和虚函数名后加final关键字。

17. i++和++i区别（效率）

当单独使用的时候，两者没有区别，参与运算时，也不过是语句的执行顺序换了。

当我们考虑自定义类的时候，就不一样了。

i++是先用临时对象保存原来的对象，然后对原对象自增，再返回临时对象，不能作为左值；++i是直接对于原对象进行自增，然后返回原对象的引用，可以作为左值。

由于要生成临时对象，i++需要调用两次拷贝构造函数与析构函数（将原对象赋给临时对象一次，临时对象以值传递方式返回一次）；

++i由于不用生成临时变量，且以引用方式返回，故没有构造与析构的开销，效率更高。

三、指针相关

1. 指针大小

在32位平台下，无论指针的类型是什么，sizeof（指针名）都是4字节，在64位平台下，无论指针的类型是什么，sizeof（指针名）都是8

```
// 指针
// 指向对象的指针
int* p = new int(0);
delete p;
// 指向数组的指针
int* p1 = new int[10];
delete[] p1;
// 指向类的指针：
string* p2 = new string;
delete p2;
// 指向指针的指针（二级指针）
int** pp = &p;
```

```
**pp = 10;
```

```
int a[3][4];
```

```
int (*p)[4]; //该语句是定义一个数组指针，指向含4个元素的一维数组
```

```
p = a; //将该二维数组的首地址赋给p，也就是a[0]或  
&a[0][0]
```

```
p++; //该语句执行过后，也就是p=p+1；p跨过行a[0]  
[]指向了行a[1] []
```

//所以数组指针也称指向一维数组的指针，亦称行指针。

//访问数组中第i行j列的一个元素，有几种操作方式：

//*(p[i]+j)、*(*(p+i)+j)、(*(p+i))[j]、p[i][j]。其中，
优先级：()>[]>*。

//这几种操作方式都是合法的。

2.野、悬空和void指针

(1)、野指针：野指针一般指在定义时未初始化的指针，指针变量存放的地址是随机的。比如：char*p,但是static关键字修饰的指针变量未初始化时默认为0，它不是野指针，比如static char *p **指针指向的位置是不可知的**（指针未初始化、指针访问越界访问、指针指向的空间被释放）

(2)、悬空指针：指针最初所指的内存被释放后，指针仍然指向原来分配给它的这块空间，这个时候该指针就是悬空指针。

无论是野指针还是悬空指针，它们都是指向了一段不安全不可控的内存空间，可能会破坏正常的数据，引发一些未知错误。

如何避免使用野指针？ 在定义指针时就对其进行初始化，如果不能确定初始化的值，可以对该指针初始化为nullptr。

如何避免使用悬空指针？ 指针所指空间被释放后，给该指针赋值为nullptr。或直接使用智能指针就不会出现空指针和野指针的情况。(因为在智能指针的构造和析构函数中有相应的处理。)

(3)、void指针:表示指针指向的数据类型不确定，在之后使用过程中强制转换它的类型。

4. 数组指针和指针数组

在32位平台下无论什么类型的指针，利用sizeof来求它的长度时，都是四个字节

在64位平台下都是8个字节。

指针数组：是数组，只是数组中存放的元素都是指针变量。

数组指针：是指针，它指向的是一个数组。

★5. 函数指针和指针函数

指针函数：是函数，只是函数的返回值是一个指针变量

函数指针：是指针，指针指向的是一个函数，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。常用于回调。定义形式如下：

```
int func(int a);  
int (*f)(int a);  
f = &func;
```

6. const和指针 常量指针和指针常量

const离变量名近就是用来修饰指针变量的，离变量名远就是用来修饰指针所指的数据，如果变量名远近都有const，就是同时修饰指针变量以及它指向的数据。

指针常量：char* const p//地址(指针本身)不可以修改，指针指向的内容可以修改。

常量指针：const char*p//指针本身可以修改(指向不同的数据)，但他们指向的数据本身不可以修改

7. this指针

每一个非静态成员函数只会诞生一份函数实例，多个同类型的对象会共用一块代码，那这块代码就需要区分是哪个对象调用的自己。this指针指向被调用的成员函数所属的对象，它是隐含在每一个非静态成员函数内的一种指针。this指针主要有两个用途：

- (1)当形参和成员变量同名时，可用this指针来区分
- (2)在类的非静态成员函数中返回对象本身，可使用return *this

8.指针和迭代器区别

迭代器实际上是一个类模板，它是对指针的封装，模拟了指针的一些功能，重载了指针的一些操作符，比如解引用*、指针指向操作符->、++、--等。

9.使用指针需要注意什么

- 1. 定义指针时，先初始化为NULL，避免“野指针”。
- 2. 用malloc或new申请内存之后，应该**立即检查**指针值是否为NULL。防止使用指针值为NULL的内存。
- 3. 不要忘记为数组和动态内存**赋初值**。防止将未被初始化的内存作为右值使用。
- 4. 避免数字或指针的下标**越界**，特别要当心发生“多1”或者“少1”操作

5. 动态内存的申请与释放必须配对，防止**内存泄漏**
6. 用free或delete释放了内存之后，立即将指针**设置为NULL**，防止“悬空指针”

10.指针和引用的区别

- 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
- 指针可以有多级，引用只有一级
- 指针可以为空，引用不能为NULL且在定义时必须初始化
- 指针在初始化后可以改变指向，而引用在初始化之后不可再改变
- sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小
- 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
- 引用本质是一个指针，同样会占4字节内存；指针是具体变量，需要占用存储空间（，具体情况还要具体分析）。
- 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
- 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
- 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

11.在传递函数参数时，什么时候该使用指针，什么时候该使用引用呢？

- 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。而返回局部变量的引用是没有意义的
- 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小
- 类对象作为参数传递的时候使用引用，这是C++类对象传递的标准方式

12.区别以下指针类型？

```
int *p[10]
int (*p)[10]
int *p(int)
int (*p)(int)
```

- `int *p[10]`表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向`int`类型的指针变量。
- `int (*p)[10]`表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个`int`类型的数组，这个数组大小是10。
- `int *p(int)`是函数声明，函数名是`p`，参数是`int`类型的，返回值是`int *`类型的。
- `int (*p)(int)`是函数指针，强调是指针，该指针指向的函数具有`int`类型参数，并且返回值是`int`类型的。

四、内存分配相关

1 C++五大分区(内存管理)

一个C/C++编译的程序占用的内存分为：堆区、栈区、全局区(静态区)、常量区、程序代码区。

栈：由编译器自动分配和释放，存放函数运行时分配的局部变量、形参、返回数据、返回地址等，它的操作类似于数据结构中的栈(是一块连续的内存)。

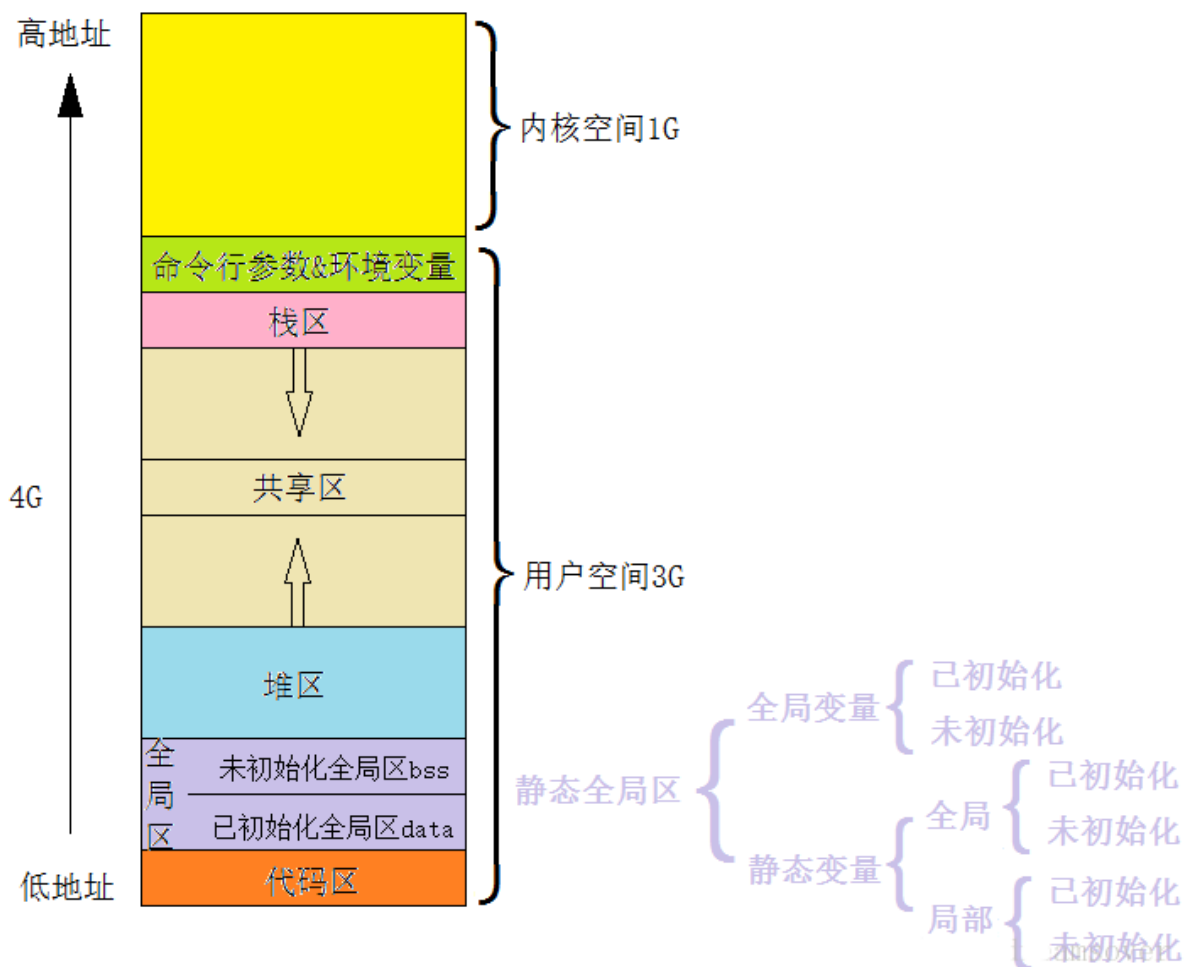
堆：堆区是动态分配，程序运行时用new或malloc来申请内存，需要程序员负责释放，如果程序员没有释放，程序结束后，操作系统会自动回收。(频繁的分配和释放不同大小的堆区空间将会造成内存碎片)

全局区(静态区)：全局区分为已初始化全局区(.data段)，存放初始化后的全局变量和静态变量，未初始化全局区(.bss段)，存放未初始化的全局变量和静态变量。这块内存有读写权限，所以全局区中的数据在程序运行期间可任意改变。这块内存是在程序编译阶段分配好的，在程序结束后会由操作系统回收。

常量区：存放一般的常量、字符串常量等，这块内存只有读权限、没有写权限，所以它们的值在程序运行期间不能改变，程序结束后由操作系统回收。

代码区：存放二进制代码。(不允许修改，但可以执行)

最后还有一个**文件映射区**，位于堆和栈之间



2.静态局部变量，全局变量，局部变量的特点，以及使用场景

1. **首先从作用域考虑**：C++里作用域可分为6种：全局，局部，类，语句，命名空间和文件作用域。

全局变量：全局作用域，可以通过extern作用于其他非定义的源文件。

静态全局变量：全局作用域+文件作用域，所以无法在其他文件中使用。

局部变量：局部作用域，比如函数的参数，函数内的局部变量等等。

静态局部变量：局部作用域，只被初始化一次，直到程序结束。

2. **从所在空间考虑**：除了局部变量在栈上外，其他都在静态存储区。因为静态变量都在静态存储区，所以下次调用函数的时候还是能取到原来的值。
3. **生命周期**：局部变量在栈上，出了作用域就回收内存；而全局变量、静态全局变量、静态局部变量都在静态存储区，直到程序结束才会回收内存。
4. **使用场景**：从它们各自特点就可以看出各自的应用场景，不再赘述。

★3.堆和栈区别

- 1.管理方式不同：栈由编译器自动申请和释放内存，堆区需要程序员手动申请和释放
- 2.碎片问题：栈区和数据结构中栈的原理一样，在弹出一个元素之前，前一元素已经弹出了，不会产生碎片(内存连续)。但堆区如果频繁的分配和释放不同大小的堆区内存会造成内存碎片。(堆区内存不连续)
- 3.生长方向不同：堆向上，向着内存地址增加的方向生长。栈向下，向着内存地址减小的方向生长。(操作系统规定的)
- 4.堆空间大，栈空间小，32位环境下虚拟地址空间大小约为4G，其中堆空间大小约为2G左右。栈空间大小约为1M。

堆快还是栈快？

毫无疑问是栈快一点。

因为操作系统会在底层对栈提供支持，会分配专门的寄存器存放栈的地址，栈的入栈出栈操作也十分简单，并且有专门的指令执行，所以栈的效率比较高也比较快。

而堆的操作是由C/C++函数库提供的，在分配堆内存的时候需要一定的算法寻找合适大小的内存。并且获取堆的内容需要两次访问，第一次访问指针，第二次根据指针保存的地址访问内存，因此堆比较慢。

4. new/delete和malloc/free

(可作为C和C++内存分配区别的一个点)

1. new和delete是操作符， malloc和free是函数。
2. 使用new创建对象会调用构造函数，使用delete释放对象时会调用析构函数， malloc和free是没有构造函数和析构函数的
3. 使用malloc需要指定申请的空间大小，还要对返回回来的void* 类型指针进行强制类型转换， new这些都不需要(会调用构造函数)
4. malloc申请完空间后要判空，因为它设定内存分配失败会返回空指针，而new申请空间后不用判空，new如果发生错误会抛出异常。(分配失败抛出异常std::bad_alloc)

	New/delete	Malloc/free 本人
本质属性	运算符	CRT 函数
内存分配大小	自动计算	手工计算
类型安全	是 (一个 int 类型指针指向 float 会报错)	不是 (malloc 类型转换成 int, 分配 double 数据类型大小的内存空间不会报错)
两者关系	new 封装了 malloc	
其他特点	除了分配和释放内存还会调用构造和析构函数	只分配和释放内存
	内存分配失败时抛出 bad_alloc 异常	内存分配失败时返回 null
	返回定义时具体类型的指针	返回的是 void 类型的指针，使用时需要进行类型转换

5.new和delete实现原理

new实现过程：第一步调用operator new()的标准库函数，申请一块足够大的内存空间，用来保存指定类型的对象；第二步，编译器运行相应的构造函数去构造这些对象，并进行初始化。第三步，对象被分配完空间并完成构造后，返回一个指向该对象的指针。

(operator new()操作的内部是调用了malloc()函数)。

delete实现过程：运行指针所指向对象的析构函数(或所指数组中的元素执行对应的析构函数)，然后通过调用operator delete(或operator delete[])的标准库函数释放所用内存。(operator delete()操作的内部是调用了free()函数)。

6. delete和delete[]的区别

delete用来释放单个对象所占的空间，只会调用一次析构函数；

delete[]用来释放数组空间，会对数组中的每个成员都调用一次析构函数。

delete[]时，数组中的元素按逆序的顺序进行销毁。

7.malloc申请内存时，OS怎么做的

malloc整体思想是先向操作系统申请一块大小适当的内存，然后自己管理，(从操作系统的角度看)，malloc是通过brk()和mmap()这两个系统调用实现的：

(1)brk是将进程数据段(.data)的最高地址指针向高处移动，通过这种方式扩大进程在运行时的堆大小。

(2)mmap是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）寻找一块空闲的虚拟内存，这样可以获得一块可以操作的堆内存。

一般分配的内存如果小于128k时，使用brk调用来获得虚拟内存，如果大于128k时使用mmap来获得虚拟内存。进程先通过这两个系统调用获取虚拟内存，获得相应的虚拟地址，然后在访问这些虚拟地址的时候，通过缺页中断，让内核分配相应的物理内存,并建立虚拟内存和物理内存之间的映射关系，完成内存分配。

malloc底层实现：当开辟的空间小于 128K 时，调用 brk () 函数；当开辟的空间大于 128K 时，调用mmap ()。malloc采用的是内存池的管理方式，以减少内存碎片。先申请大块内存作为堆区，然后将堆区分为多个内存块。当用户申请内存时，直接从堆区分配一块合适的空闲快。采用隐式链表将所有空闲块，每一个空闲块记录了一个未分配的、连续的内存地址

8.常见的内存错误及其对策

- (1) 内存分配未成功，却使用了它。
- (2) 内存分配虽然成功，但是尚未初始化就引用它。
- (3) 内存分配成功并且已经初始化，但操作越过了内存的边界。
- (4) 忘记了释放内存，造成内存泄露。
- (5) 释放了内存却继续使用它。

对策：

- (1) 定义指针时，先初始化为NULL。
- (2) 用malloc或new申请内存之后，应该**立即检查**指针值是否为NULL。防止使用指针值为NULL的内存。
- (3) 不要忘记为数组和动态内存**赋初值**。防止将未被初始化的内存作为右值使用。
- (4) 避免数字或指针的下标**越界**，特别要当心发生“多1”或者“少1”操作

(5) 动态内存的申请与释放必须配对，防止**内存泄漏**

(6) 用free或delete释放了内存之后，立即将指针**设置为NULL**，防止“野指针”

(7) 使用智能指针。

9.内存泄漏/内存溢出

内存溢出:指程序要使用的内存超过了系统实际分配的内存。

内存泄漏:是指在堆区申请的内存使用完以后没有释放，内存泄漏堆积的后果就是内存溢出。

10.C++内存泄漏

10.1 内存泄漏情况

1.使用new在堆区创建完数据以后，没有使用delete释放，造成内存泄漏

2.当用new创建了一个对象数组，释放数据时只调用了delete没有调用delete[]，这样只有数组中第一个对象执行了析构函数，数组中其他对象的内存没有回收，造成内存泄漏

3.基类的析构函数没有设置成虚函数，在实现多态的时候，如果子类有堆区数据，用delete删除指向子类的基类指针，子类的析构函数无法被调用，导致子类对象无法释放，造成内存泄漏

4.一个指针已经指向了一块内存空间，现在为这个指针重新赋值，导致该指针原来指向的那块内存空间无法找到，造成内存泄漏。

10.2 防止内存泄漏的方法

(1)养成new和delete运算符配对使用的习惯，在自由存储空间上动态分配，使用完毕后释放。

(2)使用C++提供的智能指针，可以自动完成释放资源的操作。

(3)或者借用智能指针的思想，将内存分配和释放的相关代码封装到类中，在构造的时候申请内存，析构的时候释放内存。

(4)一定要将基类的析构函数声明为虚函数

10.3 内存泄漏检查

可以使用valgrind内存检查工具，没具体用过，内存泄漏原因一般是这几种，写代码时养成好的习惯，避免这几种情况应该就不会出现内存泄漏了。

自己动手尝试过，但是复杂情况下不适用的方法，linux中，利用gdb，在调用某个函数之前设置一个断点，然后调用malloc_stats()系统函数来查看当前进程内存使用情况。在函数执行完成后(gdb下使用n，不进入函数体)，再次调用malloc_stats()系统函数查看当前进程内存使用情况，比较它们之间的差值判断是否有内存泄漏。

11.段错误发生和检测

段错误就是说访问的内存超出了系统所给这个程序的内存，一般有三种情况

(1)访问的内存可能是不存在的，或者是受系统保护的，如果用户访问到，就会发生段错误

(2)数组越界，访问到了不属于你的内存

(3)访问只读常量区的数据，并对数据进行修改。

检测方法：当一个进程发生段错误时，Linux系统会给它发送**SIGSEGV**(segmentation violation)信号，通过gdb查看调试信息文件core即可检测到段错误。

12.atomoic六种内存顺序

有六个内存顺序选项可应用于对原子类型的操作：

1. `memory_order_relaxed`：在原子类型上的操作以自由序列执行，没有任何同步关系，仅对此操作要求原子性。
2. `memory_order_consume`：`memory_order_consume`只会对其标识的对象保证该对象存储先行于那些需要加载该对象的操作。
3. `memory_order_acquire`：使用`memory_order_acquire`的原子操作，当前线程的读写操作都不能重排到此操作之前。
4. `memory_order_release`：使用`memory_order_release`的原子操作，当前线程的读写操作都不能重排到此操作之后。
5. `memory_order_acq_rel`：`memory_order_acq_rel`在此内存顺序的读-改-写操作既是获得加载又是释放操作。没有操作能够从此操作之后被重排到此操作之前，也没有操作能够从此操作之前被重排到此操作之后。
6. `memory_order_seq_cst`：`memory_order_seq_cst`比`std::memory_order_acq_rel`更为严格。
`memory_order_seq_cst`不仅是一个"获取释放"内存顺序，它还会对所有拥有此标签的内存操作建立一个单独全序。

除非你为特定的操作指定一个顺序选项，否则内存顺序选项对于所有原子类型默认都是`memory_order_seq_cst`。

13.字节对齐

字节对齐是一种空间换时间的策略，要使变量的起始地址是该变量所占内存大小的整数倍，目的是将一个数据尽量放在cpu寻址的一个步长里，避免跨步长存储，这样CPU处理数据的效率更高。

举例：一个变量占用n个字节，它的起始地址就是n的整数倍。比如一个4字节int类型变量，它的起始地址应该是可以被4整除的。如果是结构体变量，它的起始地址是其最宽数据类型成员的整数倍。

为什么要字节对齐？

字节对齐是为了提高CPU访问效率，以32位的cpu为例，它一次可以处理4字节数据(64位是8字节)，实际寻址的步长就是4个字节，也就是只对编号为4的倍数的内存进行寻址，一个int类型的变量占4个字节，如果它的地址编号为从4开始，cpu寻址一次就可以读到这个数据，如果它的地址编号从5开始，cpu就要寻址两次，第一次从编号4开始读取四个字节得到数据的前半部分，然后从编号8开始进行一次寻址，读取四个字节，得到数据后半部分，将两部分拼接一起，得到整个数据。

14.结构体字节对齐

32/64位系统各类型变量所占字节数

数据类型	32 位	64 位
char	1	1
short int	2	2
int	4	4
unsigned int	4	4
long int	4	8
float	4	4
double	8	8
long lone	8	8
char*(指针变量)	4	8

说一下结构体的对齐规则。

一、结构体对齐规则首先要看有没有用`#pragma pack`宏 [p'rægmə]声明，这个宏可以改变对齐规则，有宏定义的情况下结构体的自身宽度就是宏上规定的数值大小，所有内存都按照这个宽度去布局，`#pragma pack` 参数只能是 '1', '2', '4', '8', or '16'。

二、在没有`#pragma pack`这个宏的声明下，遵循下面三个原则：

1.结构体中元素按照定义顺序从结构体首地址开始依次置于内存中，元素会放在自身对齐大小的整数倍地址上。（也就是成员的首地址是自身大小的整数倍）这里说的地址是元素在结构体中的偏移量，结构体中首地址偏移量为0。（分配内存的顺序是按照声明的顺序）

2.基本数据类型的对齐大小为其自身的大小，结构体数据类型的对齐大小是它元素中最大对齐元素的对齐大小。（每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍）

3.结构体的总大小，应该为所有元素中最大对齐大小的整数倍，如果不满足，结构体要对齐到最大对齐大小的整数倍，填充的字节空间放置到结构体末尾（整个结构体大小必须是里面变量类型最大值的整数倍）

关于1：(char型，自身对齐值为1，short型为2，对于int,float类型，自身对齐值为4，double型，自身对齐值为8，单位字节。)

关于2：(比如结构体中有char、int、double，那结构体对齐大小就是和double一样为8，如果结构体中有数组，将数组看做是连续数个相同类型的元素即可，char name[10]，长度最长，是10，那是不是要以 10 对齐？”不是，char a[10] 的本质是10 个char 变量，所以就把它当成10个 char 变量看就行了)。

比如结构体B包含结构体A，结构体A应该从偏移量为A内最大成员整数倍开始存储。(struct B里面存放struct A，A里有char、int、double，A中最大对齐元素为double，所以A应该从8的整数倍开始存)

关于3:

```
struct A
{
    char c; //从首地址0开始，占一个字节    0-1
    double d; //放在自身对齐大小的整数倍地址上(第一个char到1，往后填充7个字节，从地址8开始，占8个字节，到16)
    8-15
    int i; //从16开始占4个字节，但还要填4个字节凑够24(结构体大小对齐到结构体最大对齐大小8的整数倍)
    16
}; //17-24
```

sizeof(A)=24而非20，三个元素占据20字节空间，三个元素中最大对齐大小为8，所以需要在结构体的尾部填充4个字节的空间凑成8的倍数，此时整个结构体的大小为24字节。

再来个例子

```
struct B
{
    int a; //占4个字节    0-3
    A a; //结构体A的对齐大小为其内部最大元素的对其大小，即double: 8，所以要从第8个位置开始对齐    8-31
}; //32
//最终结果sizeof(B) = 32
```

15. C+空结构体大小为1

为了满足C++标准规定的不同对象不能有相同地址，C++编译器保证任何类型对象大小不能为0。C++编译器会在空类或空结构体中增加一个虚设的字节（有的编译器可能不止一个），以确保不同的对象都具有不同的地址

16. 类大小计算

说明：类的大小是指类的实例化对象的大小，用sizeof对类型名操作时，结果是该类型的对象的大小。

计算原则：

(1)遵循结构体的对齐原则。

(2)与普通成员变量有关，与成员函数和静态成员无关。即普通成员函数，静态成员函数，静态数据成员，静态常量数据成员均对类的大小无影响。因为静态数据成员被类的对象共享，并不属于哪个具体的对象。

(3)虚函数对类的大小有影响，是因为虚函数表指针的影响。（虚函数表指针一般为8，直接加到

(4)虚继承对类的大小有影响，是因为虚基表指针带来的影响。

(5)空类的大小是一个特殊情况，空类的大小为 1，当用 new 来创建一个空类的对象时，为了保证不同对象的地址不同，空类也占用存储空间。

```
class A {  
    int a;  
};  
  
class B:virtual public A{  
    virtual void myfunB(){}  
};  
  
class C:virtual public A{  
    virtual void myfunC(){}  
};  
  
class D:public B,public C{  
    virtual void myfunD(){}  
};
```

//A的大小为int大小加上虚表指针大小。虚表指针为8，字节对齐后A的大小为8+8

//B, C中由于是虚继承因此大小为int大小加指向虚基类的指针的大小。

//B,C虽然加入了自己的虚函数，但是虚表指针是和基类共享的，因此不会有自己的虚表指针，他们两个共用虚基类A的虚表指针。 B/C的虚基类指针+虚表指针+A类int变量大小=8+8+8

//D由于B,C都是虚继承，因此D只包含一个A的副本，于是D大小就等于int变量的大小+B中的指向虚基类的指针+C中的指向虚基类的指针+一个A虚表指针的大小，由于字节对齐，结果为8+8+8+8=32

五、面向对象

1.简述一下什么是面向对象

1. 面向对象是一种编程思想，把一切东西看成是一个个对象，比如人、耳机、鼠标、水杯等，他们各自都有属性，比如：耳机是白色的，鼠标是黑色的，水杯是圆柱形的等等，把这些对象拥有的属性变量和操作这些属性变量的函数打包成一个类来表示

2. 面向过程和面向对象的区别

面向过程：根据业务逻辑从上到下写代码

面向对象：将数据与函数绑定到一起，进行封装，这样能够更快速的开发程序，减少了重复代码的重写过程

2.简述一下面向对象的三大特征

面向对象的三大特征是封装、继承、多态。

1. 封装：将数据和操作数据的方法进行有机结合，隐藏对象的属性和实现细节，仅对外公开接口来和对象进行交互。封装本质上是一种管理：我们如何管理兵马俑呢？比如如果什么都不管，兵马俑就被随意破坏了。那么我们首先建了一座房子把兵马俑给封装起来。但是我们目的不是全封装起来，不让别人看。所以我们开放了售票通道，可以买票突破封装在合理的监管机制下进去参观。类也是一样，不想给别人看到的，我们使用

protected/private把成员封装起来。开放一些共有的成员函数对成员合理的访问。所以封装本质是一种管理。

2. 继承：可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

三种继承方式

继承方式	private继承	protected继承	public继承
基类的private成员	不可见	不可见	不可见
基类的protected成员	变为private成员	仍为protected成员	仍为protected成员
基类的public成员	变为private成员	变为protected成员	仍为public成员

3. 多态：用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。实现多态，有二种方式，重写，重载。

3.简述一下 C++ 的重载和重写，以及它们的区别

1. 重写

是指派生类中存在重新定义的函数。其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致。只有函数体不同（花括号内），派生类对象调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有virtual修饰。

示例如下：

```
#include<bits/stdc++.h>

using namespace std;

class A
{
public:
    virtual void fun()
    {
        cout << "A";
    }
};

class B :public A
{
public:
    virtual void fun()
    {
        cout << "B";
    }
};

int main(void)
{
    A* a = new B();
    a->fun(); //输出B，A类中的fun在B类中重写
}
```

2. 重载

我们在平时写代码中会用到几个函数但是他们的实现功能相同，但是有些细节却不同。例如：交换两个数的值其中包括 (int, float,char,double)这些个类型。在C语言中我们是利用不同的函数名来加以区分。这样的代码不美观而且给程序猿也带来了很多的不便。于是在C++中人们提出了用一个函数名定义多个函数，也就是所谓的函数重载。函数重载是指同一可访问区内被声明的几个具有

不同参数列（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。

```
#include<bits/stdc++.h>

using namespace std;

class A
{
    void fun() {};
    void fun(int i) {};
    void fun(int i, int j) {};
    void fun1(int i,int j){};
};
```

4.说说C++ 的重载和重写是如何实现的

1. C++利用命名倾轧（name mangling）技术，来改名函数名，区分参数不同的同名函数。命名倾轧是在编译阶段完成的。

C++定义同名重载函数：

```
#include<iostream>
using namespace std;
int func(int a,double b)
{
    return ((a)+(b));
}
int func(double a,float b)
{
    return ((a)+(b));
}
int func(float a,int b)
{
    return ((a)+(b));
}
int main()
```

```
{  
    return 0;  
}
```

2. 在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

1. 用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。
2. 存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
3. 多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。
4. 重写用虚函数来实现，结合动态绑定。
5. 纯虚函数是虚函数再加上 = 0。
6. 抽象类是指包括至少一个纯虚函数的类。

纯虚函数：virtual void fun()=0。即抽象类必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

5.说说 C 语言如何实现 C++ 语言中的重载

参考答案

c语言中不允许有同名函数，因为编译时函数命名是一样的，不像c++会添加参数类型和返回类型作为函数编译后的名称，进而实现重载。如果要用c语言显现函数重载，可通过以下方式来实现：

1. 使用函数指针来实现，重载的函数不能使用同名称，只是类似的实现了函数重载功能
2. 重载函数使用可变参数，方式如打开文件open函数

3. gcc有内置函数，程序使用编译函数可以实现函数重载

示例如下：

```
#include<stdio.h>

void func_int(void * a)
{
    printf("%d\n",*(int*)a); //输出int类型，注意 void
    * 转化为int
}

void func_double(void * b)
{
    printf("%.2f\n",*(double*)b);
}

typedef void (*ptr)(void *); //typedef申明一个函数指针

void c_func(ptr p,void *param)
{
    p(param); //调用对应函数
}

int main()
{
    int a = 23;
    double b = 23.23;
    c_func(func_int,&a);
    c_func(func_double,&b);
    return 0;
}
```

6.说说构造函数有几种，分别什么作用

参考答案

C++中的构造函数可以分为4类：默认构造函数、初始化构造函数、拷贝构造函数、移动构造函数。

1. 默认构造函数和初始化构造函数。 在定义类的对象的时候，完成对象的初始化工作。

```
class Student
{
public:
    //默认构造函数
    Student()
    {
        num=1001;
        age=18;
    }
    //初始化构造函数 （使用初始化列表）
    Student(int n,int a):num(n),age(a){}
private:
    int num;
    int age;
};
int main()
{
    //用默认构造函数初始化对象s1
    Student s1;
    //用初始化构造函数初始化对象s2
    Student s2(1002,18);
    return 0;
}
```

有了有参的构造了，编译器就不提供默认的构造函数。

2. 拷贝构造函数

```
#include "stdafx.h"
#include "iostream.h"
```

```

class Test
{
    int i;
    int *p;
public:
    Test(int ai,int value)
    {
        i = ai;
        p = new int(value);
    }
    ~Test()
    {
        delete p;
    }
    Test(const Test& t)
    {
        this->i = t.i;
        this->p = new int(*t.p);
    }
};
//复制构造函数用于复制本类的对象
int main(int argc, char* argv[])
{
    Test t1(1,2);
    Test t2(t1); //将对象t1复制给t2。注意复制和赋值的概念不同
    return 0;
}

```

赋值构造函数默认实现的是值拷贝（浅拷贝）。

3. 移动构造函数。用于将其他类型的变量，隐式转换为本类对象。下面的转换构造函数，将int类型的r转换为Student类型的对象，对象的age为r，num为1004.

```
Student(int r)
{
    int num=1004;
    int age= r;
}
```

7.说说一个类，默认会生成哪些函数

参考答案

定义一个空类

```
class Empty
{
};
```

默认会生成以下几个函数

1. 无参的构造函数

在定义类的对象的时候，完成对象的初始化工作。

```
Empty()
{
}
```

2. 拷贝构造函数

拷贝构造函数用于复制本类的对象

```
Empty(const Empty& copy)
{
}
```

3. 赋值运算符

```
Empty& operator = (const Empty& copy)
{
}
```

4. 析构函数（非虚）

```
~Empty()
{
}
```

六、多态与虚函数

1. C++多态原理

(C++虚函数原理/讲讲虚函数运行机制/讲讲虚函数如何解析)

一个类中如果包含了一个虚函数，在编译时会生成一个虚表，子类继承了父类的虚函数，每个子类也会生成一个自己的虚表，这样子类和父类在创建对象时，在对象的内存模型中，都会有一个虚表指针，指向类的虚表，类的所有对象共享这一个虚表，但类的每个对象都有一个属于自己的虚表指针，基类指针指向不同的对象去调用非静态成员函数时，会把这个对象的地址也就是this指针传递给对象的虚表指针，到虚表中找到对应的虚函数入口地址，然后执行实际被调用的虚函数。

实现多态的三个条件：1.必须有继承关系2.基类中必须包含虚函数，派生类一定要对基类中的虚函数进行重写3.通过基类对象的指针或引用指向调用虚函数

生成子类虚函数表需要三个步骤：

第一步，将父类虚函数表内容拷贝到子类虚函数表上；

第二步，将子类重写的虚函数覆盖掉表中父类的虚函数；

第三步，如果子类有新增加的虚函数，按声明顺序加到最后。

2.虚函数表原理

虚函数表就像一个数组，数组里每个位置都存放着一个指向虚函数的指针(虚函数入口地址)，在有虚函数的类的对象中，这个表被分配在这个对象的内存里面，当我们用父类的指针来操作一个子类的时候，这张表就像一个地图一样，指明实际应该调用的函数。

(每个类使用一个虚函数表，每个类对象用一个虚表指针)

3. 重载、重写和隐藏

1.重载

同一作用域下的同名函数，才存在重载关系。它的特点是函数名相同，参数的类型、数量或者顺序有所不同，函数的返回值不能够作为函数重载的条件。

下列运算符不可以重载：

- (1) "." (类成员访问运算符)
- (2) ".*" (类成员指针访问运算符)
- (3) "::" (域运算符)
- (4) "sizeof" (长度运算符)
- (5) "?:" (条件运算符)

返回值不作为重载条件：

C++编译器和链接器通过函数签名来识别不同的函数，函数签名里面包括函数名、参数类型、参数个数、顺序，还有它所在的类和命名空间。普通函数签名并不包含函数返回值部分，所以如果两个函数只有函数返回值不同，系统肯定没法区分这两个函数。(重载原理：编译器在编译.cpp文件中当前使用的作用域里的同名函数时，根据函数形参的类型和顺序会对函数进行重命名,当发生函数调用时，编译器去匹配重命名后的函数，整个过程和返回值无关)

2.重写

重写就是在派生类中重写基类中的虚函数，重写函数体。派生类重写的函数要与基类虚函数的参数类型、参数个数、返回值类型都要相同。

重载与重写区别：

- (1).重写是父类与子类之间的垂直关系，重载是不同函数之间的水平关系。
- (2).重写要求函数的参数列表要相同，重载要求函数的参数列表要不同。
- (3).重写是根据父类指针指向的对象类型来决定调用哪个函数，重载是根据函数的参数列表来决定调用哪个函数。

3.隐藏(阿秀33)

指的是在某些情况下，派生类中的函数屏蔽了基类中的同名函数：

比如：(1)派生类和基类中的两个函数参数相同，但是基类函数不是虚函数，在类外调用这个同名函数时，默认调用派生类中的函数，**和重写的区别在于基类函数是否为虚函数**

(2)两个函数参数不同，无论基类函数是否为虚函数，都会被隐藏。如果想重载成员函数，这两个函数必须在一个类中，**和重载的区别在于两个函数不在同一个类中。**

4.虚析构、不要虚构造

为什么不要虚构造/构造函数和析构函数中能否调用虚函数，为什么？/基类的析构函数为什么通常定义为虚函数

1.要虚析构

在实现多态时，用基类指针指向派生类对象，如果基类析构函数不是虚函数，并且子类在堆区创建了数据，用delete删除指向派生类的基类指针时，就只会调用基类的析构函数，不会调用派生类的析构函数，导致派生类堆区数据无法释放，造成内存泄漏，所以为了避免这种情况就要使基类析构函数为虚函数。(为什么只会调用基类析构，不调用派生类析构，很多博客说的是：因为析构重名,只能调用一个,调用默认的父亲析构函数，可是基类和派生类析构函数名不一样啊)

2.不要虚构造

若构造函数为虚函数，说明函数的调用方式是通过虚函数表来调用的，但是虚函数表是在构造函数调用后创建完对象才存在的，这就出现了一种类似死锁的情况，所以构造函数不可能为虚函数。

(虚函数的作用是为了通过父类的指针或者引用来调用它的成员函数时能够变成调用子类的成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用)

5.哪些函数不能是虚函数

(1)内联函数：内联函数在编译阶段把该函数代码副本放在每个函数调用的地方，所以它是静态行为(编译时)，而虚函数是动态行为(运行时)。

(2)构造函数：若构造函数是虚函数，说明函数的调用方式是通过虚函数表，而虚函数表是在构造函数创建完对象以后才存在的，所以构造函数不可能为虚函数。

(3)静态成员函数：因为虚函数的调用关系是，指向当前对象的this指针调用对象的虚表指针，通过虚表指针在虚表中找到虚函数入口地址，执行虚函数，静态成员函数是一个类共享的函数，不属于某一个对象，不能用this指针来访问它，所以如果把它设置为虚函数在虚函数表中根本找不到。

6.虚与纯虚函数(区别)

被virtual关键字修饰的成员函数就是虚函数，C++中使用虚函数是为了实现多态，在基类中定义虚函数，在派生类中重写这个虚函数，真正执行的虚函数是引用所绑定或指针所指向的对象所属的成员函数。

纯虚函数是在基类中虚函数的原型后面写=0，有的时候，基类不能对虚函数给出有意义的实现，就把它声明为纯虚函数，它的实现留给该基类的派生类去做，这就是纯虚函数的作用。

虚函数和纯虚函数的区别：

- 1.定义形式不同:虚函数在定义时是在普通函数基础上加上virtual关键字，纯虚函数定义时除了加上virtual关键字还需要加上=0;
- 2.包含纯虚函数的类是抽象类，不能实例化对象，而包含虚函数的类可以实例化对象。
- 2.纯虚函数一般没有代码实现部分，而虚函数一般必须要有代码实现部分，否则会出现函数未定义错误。

7.引入纯虚函数目的

因为会出现在基类中不能对虚函数给出有意义的实现的情况，这时候把虚函数声明为纯虚函数，纯虚函数的代码实现留给该基类的派生类去做，这就是纯虚函数的作用。

8.虚表生成，虚表指针赋值

虚函数表在程序编译时生成，虚表指针是在非静态成员函数调用时被赋值的，非静态成员函数被调用时会传入一个隐藏的参数，这个参数是this指针，它是该对象的地址，把它赋值给虚表指针。

9.虚表底层布局

看不懂，不清楚，太难了，应该不会问到吧。

10.子对父类的非虚函数重写，通过基类指针调用的函数是哪版？

是父类的版本，因为通过指针访问非虚函数，编译器会根据指针的类型来确定要调用的函数，也就是指针指向哪个类，就调用哪个类的成员函数。

而如果指针访问的是虚函数，并且派生类重写了该虚函数，编译器会根据指针指向的对象找到被调用的函数，也就是指针指向的对象属于哪个类，就调用哪个类的成员函数。

整个过程应该是，包含虚函数的类，在编译阶段就生成了一个虚函数表，然后这个类在创建对象时，对象的内存模型中会有一个虚表指针，

11.派生类有两个基类，每个基都有虚函数，派生类对象内存中有几个虚表指针

两个，派生类拥有多少个虚函数基类，派生类对象就有多少个虚表指针。

且派生类的虚函数的地址跟第一基类的虚函数地址保存在同一张虚函数表中(没明白什么意思，派生类和基类不是都各自有各自的虚表吗？派生类那张表里不应该既有继承到的第一基类的虚函数地址也有第二基类的虚函数地址)



14

12.一个对象访问普通成员函数和虚函数哪个更快？

- 1.如果是通过普通对象进行访问的，访问效率是一样的。
- 2.如果是通过指针或引用进行访问的，访问普通函数更快，因为通过指针或引用访问虚函数需要运行时查询虚函数表才能确定要调用的函数地址，而访问普通函数，在编译时就可以确定要调用的函数地址。

13.虚表、虚表指针、虚函数之间的关系？以及它们都存放在哪？

虚函数和普通函数一样存放在代码区，虚表指针存在于对象中，虚表存在于全局区。

虚表指针是存在于对象中，虚表指针指向虚表，即虚表指针就是虚表的首地址。虚表是一个函数指针数组，虚表中存放的都是类中的虚函数的地址。实现多态的时候，通过虚表指针，找到虚表，在通过虚表找到虚函数的地址，调用虚函数。

七.类和对象

1.构造和析构函数

- 构造函数：在创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。
- 析构函数：在对象**销毁前**系统自动调用，执行一些清理工作。

2.C++类中默认成员函数

- 1.默认构造函数(无参、函数体为空)
- 2.默认析构函数(无参、函数体为空)
- 3.默认拷贝构造函数，对属性进行值拷贝(注意是值拷贝)
- 4.赋值运算符重载(赋值运算符重载必须有返回值，否则无法解决连续赋值操作)
- 5.取地址运算符重载

3.初始化列表

初始化列表好处：(1)统一初始化方式，直接在变量名后面跟上初始化列表，来进行对象的初始化。

(2)如果构造函数的参数中有自定义的类型，使用初始化列表，会直接调用该成员变量对应的构造函数即完成初始化。如果是在构造函数中初始化，对象的成员变量会先调用默认构造函数为成员变量初始化，然后再调用该成员变量对应的构造函数再初始化一次。

有些类型的数据无法通过在构造函数体内进行赋值来进行初始化，它们必须使用初始化列表：

- 1.常量成员(const)，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
- 2.引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面

3.自定义类型成员且没有默认构造函数的类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

4.拷贝构造的调用时机

- 1.当用类的一个对象去初始化该类的另一个对象时；
- 2.如果函数的形参是类的对象，调用函数进行形参和实参结合的时候；
- 3.如果函数的返回值是类对象，函数调用完成返回的时候。

5.深浅拷贝，如何实现深拷贝？

什么时候需要自定义拷贝构造函数？

浅拷贝：就是简单的等号赋值，将源对象的值拷贝到目标对象中，新旧对象共享同一块内存(这也是浅拷贝会发生重复释放同一块堆区内存的原因)。

深拷贝：是在堆区申请一块和源对象同样大小的内存空间，再将源对象中的数据拷贝到目标对象中，这样源对象和目标对象分别独立的占用一块内存空间，并且里面的内容还是一样的，当指向这两块内存的指针先后去调用析构函数时，分别释放自己所指空间的数据，这样就不会出现堆区内存重复释放的问题。

什么时候使用深拷贝

对于含有指针成员的对象，调用拷贝构造函数对其进行初始化时，为了防止出现堆区内存重复释放的问题，就需要使用深拷贝的方式


八.组合和继承

7.1什么是组合和继承

组合和继承是面向对象中两种代码复用的方式。组合是在新类中创建原有类的对象，也可以调用原有类的成员函数，重复利用已有类的功能，继承是创建一个新类，直接获取到现有类的成员，在这个基础上在定义新类。

如果每个A类对象是一个B类对象会出现继承关系(IS-a)，A类中包含B类对象作为其成员会出现组合关系(Has-a)。

```
template <class T>
class queue {
    ...
protected:
    deque<T> c;      // 底層容器
public:
    // 以下完全利用 c 的操作函數完成
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    reference back() { return c.back(); }
    //
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```



7.2组合与继承的区别/对比

组合关系相比于继承关系，它的整体类和局部类之间相对独立，整体类的实现细节不会暴露给局部类，封装性好，且支持动态扩展，在运行时根据具体对象选择不同类型的对象进行组合。而继承关系中父类实现细节会暴露给子类，子类会依赖父类，二者耦合度高，修改父类代码时也要对子类修改，增加维护难度，同时子类不支持动态扩展，在编译时就决定了父类。

